Python File Reading

Python makes it easy to read the data out of a text file. There are a few different forms, depending on if you want to process the file line by line or all at once.

Here is the canonical code to open a file, read all the lines out of it, handling one line at a time.

```
with open(filename, 'r') as f:
    for line in f:
        # look at line in loop
        print(line, end='')
```

The `'r'` here indicates reading a file from the filesystem into memory. (The mode `'w'` is for file writing, shown below.) The `'r'` is actually the default, so calling open() without mentioning any letter does reading, so this is the same as the above form:

```
with open(filename) as f:
    ...
```

When reading lines out of a file, each line has a '\n' char at its end. The lines from the file are fundamentally **text**. Use functions like int() to convert text to int:

```
>>> line = '123\n'   # line is textual
>>> int(line)        # Compute int value
123
```

Use s.split() with a parameter to separate one line into parts, like this:

```
>>> line = 'apple,12,donut\n'
>>> line.split(',')  # note ',' param
['apple', '12', 'donut\n']
```

Use s.strip() to remove whitespace

```
>>> line = '   this   \n'
>>> line.strip()
'this'
```

The advantage of processing 1 line at a time is that it does not require memory to hold every byte of the file at once. It's not uncommon to have a text file with millions of lines of data. With this form, only one line at a time must be stored in RAM, not all of the lines at once.

Other Ways To Read A File

Suppose we have this 2 line file:

```
Roses are red
Violets are blue
```

2. text = f.read()

(Can try these in >>> Interpreter, running Python3 in a folder that has a text file in it we can read, such as the "wordcount" folder.)

Read the whole file into 1 string - less code and bother than going line by line. Handy if the code does not need to go line by line. Can use with split() to process the whole file at once. This will require memory in Python to store all of the bytes of the file. As an estimate, look at the byte size of the file in your operating system file viewer.

```
with open(filename, 'r') as f:
    text = f.read()
    # Look at text str
```

In this example text is the string `'Roses are red\nViolets are blue\n'`

The read() function is designed to be called **once**, and it returns the entire contents of the file. Do not call read() a second time; store the text string returned by the first call and use that.

Recall the function s.split() with no parameters, splits on whitespace, returning a list of "words". Whitespace includes '\n', and the no-param form of split merges multiple whitespace chars together.

Therefore, split() works nicely with the whole text of a file, treating '\n' like just another whitespace char:

```
text = 'Roses are red\nViolets are blue\n'
>>> text.split()
['Roses', 'are', 'red', 'Violets', 'are', 'blue']
```

So text = f.read() may be followed by a words = text.split(). Now we have a list of words easily, and we do not bother with lines or looping.

Demo - read the whole book into 1 string, split into words. Python looks powerful here.

```
>>> with open('alice-book.txt', 'r') as f:
...   text = f.read()
>>> len(text)   # num chars,  len > 149,000 !
149103
>>>
>>> text[:200]
"Alice's Adventures in Wonderland\n\n                ALICE'S ADVENTURES IN
WONDERLAND\n\n                                  Lewis Carroll\n\n              THE
MILLENNIUM FULCRUM EDITION 3.0\n\n\n\n\n\n                                  "
>>>
>>> words = text.split()
>>> len(words)  # num words
26963
>>> words[:20]
["Alice's", 'Adventures', 'in', 'Wonderland', "ALICE'S", 'ADVENTURES', 'IN',
'WONDERLAND', 'Lewis', 'Carroll', 'THE', 'MILLENNIUM', 'FULCRUM', 'EDITION', '3.0',
'CHAPTER', 'I', 'Down', 'the', 'Rabbit-Hole']
```

Conclusion: 3 lines of Python, can just have a list of all the words, ready for a loop or whatever.

3. lines = f.readlines()

f.readlines() returns a list of strings, 1 for each line. Sometimes it's more useful to have all the lines at once, vs getting them 1 at a time in the standard loop. Can slice etc. to control which lines we access.

```
with open(filename, 'r') as f:
    lines = f.readlines()
    # use lines list lines[0], lines[1], ..
```

Here lines is `['Roses are red\n', 'Violets are blue\n']`. The lines list is analogous to the for-line-in-f loop, but in the form of a list.

What The "With" Does

The `with open(...)` form automates closing the file reference when the code is done using it. Closing the file frees up some memory resources associated with keeping the file open. In older versions of Python (and in other languages) the programmer is supposed to call `f.close()` manually when done with the file. Here is an example of file reading written the old way:

```
# old way to do it, call f.close() manually
f = open(filename, 'r')
...use f...
f.close()
```

Nowadays, using the `with open(..)` structure, code can concentrate on reading and the closing is automatic and we don't have to think about it.

File Writing

File "writing" is the opposite direction of reading - taking data in Python and writing it out to a text file. The CS106a projects typically do lots of reading, which is the most common form.

Here is example code writing to file (and you can try this in the interpreter). First specify `'w'` in the open(). Then call `print('Hello', file=f)` to print data out to the file as a series of text lines. This is the same `print()` function that writes to standard output, here used to write to the opened file.

```
>>> with open('out.txt', 'w') as f:
...   print('Hello there', file=f)
...   print('Opposite or reading!', file=f)
```

After running those lines, a file `out.txt` now exists in the directory from which we ran Python:

```
$ cat out.txt
Hello there
Opposite or reading!
```